Minimizing Exascale Memory Bandwidth Usage through Sparse Matrix Compression

Jeremiah Willcock, jewillco@crest.iu.edu *Andrew Lumsdaine*, lums@crest.iu.edu

Center for Research in Extreme Scale Technologies, Indiana University

Abstract

Although system memory bandwidths are high and increasing in absolute terms, they are not growing nearly as fast as available FLOPS rates. Thus, memory bandwidth per core and per FLOP are shrinking; exascale systems are expected to have this ratio far lower than today's systems. As sparse matrices, often from meshes, are important in many scientific applications, and matrix-vector multiplication is a common operation on them, moving matrix data can consume a substantial part of a system's available memory bandwidth. Thus, sparse matrix compression, both of index data and of values, will provide important performance and power benefits at exascale and should be adopted by applications.

Scientific applications, especially those that involve solving partial differential equations, often include linear algebra operations on sparse matrices such as matrix-vector multiplication, solution of linear equations, and eigenvalue problems. For the latter two of these operations, iterative methods are often used for performance; these methods use matrix-vector multiplication as a fundamental operation. Thus, the performance of these multiplication operations is often important to overall application performance.

In traditional implementations of iterative methods, matrix-vector multiplications are done one-at-a-time, with dependencies between successive multiplications that prevent their combination. Although work has been done to alleviate this issue (e.g., the algorithms in [2] reuse matrix data across several multiplications even though that work does not consider memory bandwidth issues explicitly), these approaches require modifications to the underlying mathematical algorithms. For a single matrix-vector multiplication (or a matrix-multiple vector multiplication used in some algorithms), the data from the matrix is only used once. In typical applications, sparse matrices are large enough that they do not fit into even the last level of system cache, and thus no data reuse is possible. Because optimizations (both hardware and software) can hide the latency of vector accesses and increase reuse for those, matrix data can quickly become a bottleneck. Unlike latency, bandwidth limitations cannot be hidden by multiple threads or prefetching. This constraint has been previously identified and studied by Gropp et al [3], and it is expected to be worse at exascale; according to the 2008 DARPA Exascale Study:

> *Finally, the increase in main memory latency and decrease in main memory bandwidth relative to processor cycle time and execution rate continues. This trend makes memory bandwidth and latency the performance-limiting factor for many applications, and often results in sustained application performance that is only a few percent of peak performance. [4, p. 27]*

Memory traffic, along with other data transfers, is also expected to consume a substantial part of an exascale system's power budget:

> *For example, given that a DDR3 chip today consumes just over 600 mW/GB/sec, the power budget for an Exascale data center machine requiring 1 EB/sec of main memory bandwidth would be just over 600 megawatts, which is simply not viable. [4, p. 119]*

The memory bandwidth constraint on matrix-vector multiplication performance for a square matrix can be expressed using the following formula:

$$T \geq \frac{N_{index} + N_{value} + 2sn}{bandwidth} \quad (1)$$

In the formula, T is the total multiplication time in seconds, N_{index} is the number of bytes of matrix index data, N_{value} is the number of bytes of matrix value data, s is the size of a single matrix or vector element, n is the dimension of the matrix, and *bandwidth* is the memory bandwidth in bytes per second. This formula is idealized: it assumes that there is perfect reuse of the vector data, and that neither the matrix nor vectors start in cache. It thus provides a lower bound on the time for a single multiplication operation. The formula shows that reducing the size of the

matrix index and/or value data will have a direct benefit for achievable performance. For a compressed sparse row matrix using double-precision values and 32-bit integers, the formula can be simplified to the following, based on calculations in [6]:

$$T \geq \frac{12nnz + 20n + 4}{bandwidth} \tag{2}$$

Even with suboptimal reuse of vectors leading to increases in the coefficient of n, this formula gives a real constraint since nnz is much greater than n in practice. Therefore, compressing the matrix index and/or value data provides a performance benefit, as long as the data can be decompressed on the fly during the matrix-vector multiplication. Using the idealized formula, just compressing the index data by a ratio of $1-r$, as in [6], reduces the time bound to:

$$T \geq \frac{(4r+8)nnz + (4r+16)n + 4r}{bandwidth} \tag{3}$$

Compressing the matrix's data values (as done, for example, by [5]) or the vectors provides an even greater improvement. Several authors have showed practical improvements on real systems based on these techniques. For example, Willcock and Lumsdaine [6] compress the index data of a variety of representative matrices by factors of 40–90%, leading to performance improvements up to 30%; the actual performance of one of their algorithms is shown in Figure 1. Even with the time of compression and decompression included, relatively few operations with the same matrix pattern are required to provide a net benefit. The work in [5] evaluates index and value compression on multi-core systems, using simpler compression algorithms that compress less aggressively but can be optimized more effectively for decompression and multiplication performance. The value compression in that work looks for repeated coefficients in the matrix and stores only unique values, leading to greater compression and performance improvement for some matrices.

Preconditioning is often considered important to the performance of iterative methods for sparse linear algebra. Several types of preconditioners, including incomplete LU and Cholesky factorizations and sparse approximate inverse approaches, use the input sparse matrix's pattern or modifications of it as the pattern of the preconditioning matrix (or its factors). These methods then use matrix-vector multiplication

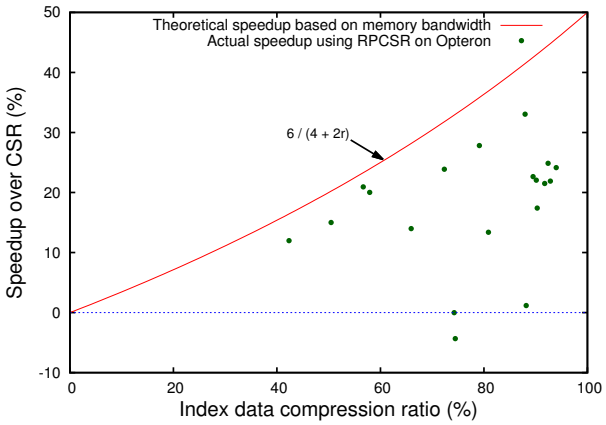

Fig. 1: Performance using one compression algorithm from [6]; green dots represent individual matrices from the Matrix Market and University of Florida collection [1].

or triangular solve operations to apply the preconditioner. Both of these operations have fundamentally the same issue as the original matrix-vector multiplication problem: a large matrix is processed against a vector and the matrix data is too large to remain in cache across multiple operations. Thus, compression algorithms help in this case as well.

Compression is likely to provide further benefit in the future. As bandwidth limitations become more stringent, compression will become more effective for performance. The increasing flexibility of bitwise and SIMD operations on modern and upcoming processors is likely to allow better compression ratios to be possible while still achieving high decompression performance; branches also have lower latency penalties that are easier to hide with multiple threads. Some work has already taken advantage of limited matrix bandwidths to reduce index data sizes on GPUs [7]. Taking advantage of information on the sources of the matrices being processed (such as that they are finite element matrices assembled from a mesh) can also provide benefit, even when methods still use fully assembled, explicitly stored matrices. Finally, time skewing and similar methods allow matrix data to be reused across several matrix-vector multiplications (with adjustments to the mathematical analyses and structures of the iterative algorithms themselves), but matrix transfer speeds are still a performance bottleneck even with those improvements [2].

References

[1] Timothy A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[2] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*, pages 1–12, 2008. http://parlab.eecs.berkeley.edu/sites/all/parlab/files/Demmel.pdf.

[3] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer, editor, *Parallel CFD*. Elsevier, 1999. http://citeseer.ist.psu.edu/gropp99towards.html.

[4] Peter M. Kogge (ed.). ExaScale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, University of Notre Dame, September 2008. http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf.

[5] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Exploiting compression opportunities to improve spmxv performance on shared memory systems. *ACM Trans. Archit. Code Optim.*, 7(3):1–31, December 2010.

[6] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *International Conference on Supercomputing*, pages 307–316, June 2006.

[7] Shiming Xu, Wei Xue, and Hai Xiang Lin. Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform. *The Journal of Supercomputing*, 63(3):710–721, 2013.